Southern Software

# ACCEL3/4

## Compiler for TRS-80® BASIC

southern software

ALGORIX Box 11721 San Francisco CA 94101

PO Box 39, Eastleigh, Hants, England, SO5 5WQ

CONTENTS Page



In this manual ACCEL3/4 means either ACCEL3 or ACCEL4. It is assumed that ACCEL4 is the preferred compiler. You can use ACCEL4 directly from the product disk, or better from a BACKUP of the product disk, or you can COPY the modules ACCEL/CMD, ACCP1/CMD, ACCP2/CMD, and ACCRT/CMD to your own system disk using the IMPORT utility supplied. If you need to use ACCEL3 (to run on another operating system other than TRSDOS, LDOS, or smal-LDOS), then start by reading the section of the manual entitled INSTALLING ACCEL3.



## CONTENTS OF PRODUCT DISKETTE

The product disk is either 35-track (for TRS-80 Model I, Video Genie, or PMC-80), or it is 40-track (for TRS-80 Model III). Or it may be double-sided (35-track on the labelled side, 40-track on the reverse). Depending on your source of supply, the disk may contain a full operating system, or it may be a "DATA" disk, containing MINOS, the Southern Software MINimum Operating System, which will allow it to boot up on drive 0.

If you have multiple disk drives, or an Operating System that allows full copying on a one-drive system, then treat the product disk as a DATA disk, and COPY over the files you need. But if you have TRSDOS, and a single-drive system, then you will need to use the IMPORT utility which is provided on the product disk. Boot up the product disk, and obey the indicated sequence of two swaps between it and your TRSDOS disk. This will install IMPORT/CMD on to your TRSDOS disk. Now type IMPORT, and, when instructed, reinsert the product disk. You will get a full-screen display showing all the files on the disk, and you can copy over those you need to your system disk (via memory).

The programs are:

| | |
|---|---|
| ACCEL/CMD | These four are the ACCEL4 product |
| ACCP1/CMD | |
| ACCP2/CMD | |
| ACCRT/CMD | |
| ACCEL3/CMD | This is the ACCEL3 product |
| ACC3RT/CMD | Optional run-only ACCEL3 component |
| ACCSU/CMD | Set-up module, to allow compiled programs to run from DOS |
| IMPORT/CMD | Import utility (see above) |
| RELOCATE/CMD | Relocation utility (see INSTALLING ACCEL3) |
| MUSIC/BAS | Sample programs |
| OVERRUN/BAS | |
| MICE/BAS | |

## A SAMPLE COMPILATION

ACCEL3/4 is very easy to use. Enter the following sample program in the normal way, just as if you were developing any BASIC program for the TRS-80.

```
10 'SAMPLE
20 DEFINT I-J
30 FOR I=1 TO 1000:NEXT
40 A$ = A$ + "X"
50 PRINT J; A$;
60 J = J + 1
70 IF J<5 THEN 30
80 STOP
```

List the program, check it, run it, and change it, if necessary. Once you have compiled it you will no longer be able to EDIT it. So SAVE it.

To use ACCEL3/4 "from cold", make sure you have a disk mounted which contains the ACCEL4 modules supplied on the product disk, (or the module ACCEL3/CMD that you built by installing ACCEL3) and type:

```
CMD"I","ACCEL" (enter)     or CMD"I","ACCEL3" (enter)

The computer will reply with:
ACCEL4 (C) COPYRIGHT SOUTHERN SOFTWARE 1982
116 98 347     (These three values are the changing program size, slightly different for ACCEL3)
READY
```

Your BASIC program has now been irreversibly converted to machine-code. You cant EDIT it in any way, but you can LIST it. Shown in comparison with the original, it will look like this:

| Before Compilation | Compiled by ACCEL3/4 |
|---|---|
| 10 'SAMPLE | 10 : |
| 20 DEFINT I-J | 20 DEFINTI-J: |
| 30 FOR I=1 TO 1000:NEXT | |
| 40 A$ = A$ + "X" | |
| 50 PRINT J; A$; | |
| 60 J = J + 1 | |
| 70 IF J<5 THEN 30 | |
| 80 STOP | 80 STOP |

1) Lines in the program that have been converted to machine-code do not appear in the listing. (The actual machine-code itself follows the dangling :, but is unprintable).

2) I and J were defined as INTEGERs in line 20, and as a result the machine-code compiled will be much faster than if they had been float variables (SINGLE or DOUBLE).

3) DEFINT, and STOP were not compiled, but the run-time environment is smart enough to ensure that the BASIC interpreter is passed control for these statements, and that its understanding of any variables they refer to is the same as that of the compiled code.

## RUNNING A COMPILED PROGRAM

```
RUN (enter)
0 X 1 XX 2 XXX 3 XXXX 4 XXXXX (program runs)
BREAK IN 80
READY
```

A second RUN will rerun the program. GOTO 10 or GOTO 20 will reenter the program without resetting J to 0 or A$ to null. GOTO 30, or a reference to any of the lines that have disappeared will result in an UNDEFINED LINE NUMBER message. RUN it again, but hit BREAK to interrupt the program before completion. Note that this throws you into READY, without the BREAK IN N message. Type ?I;J;A$ to interrogate the current values of the variables. CONT will not work after BREAK. In a larger program the BREAK key may arbitrarily "take" in a compiled line, or in an interpreted line. In the latter case, CONT will work. In either case the variable values are correct. Type J=2, and then GOTO 10 to restart execution, with a modified value of J. Turn trace on by typing TRON, and rerun the program. Only the uncompiled lines are traced. Turn trace off again with TROFF.

Once you have compiled a program, you can no longer use the commands EDIT, AUTO, DELETE, NAME (renumber), or MERGE. This is because the machine-code in the compiled lines may contain bytes that are treated as control codes by the interpreter. So use of these commands may cause an infinite loop, or a machine reboot. To get the machine back to its normal, editable state, you must use NEW, CLOAD, LOAD or RUN "program-name". All of these destroy the compiled program. Reload your saved SAMPLE program, relist and rerun it. To compile a second or subsequent program, you must have the ACCEL3/4 modules accessible on disk as before, but you can use the shorthand command: /FIX (enter)

This is identical in effect to CMD"I","ACCEL", but much easier to type. You need only use CMD"I","ACCEL" for the first compilation of a session (or after any subsequent reinitialisation of DISK BASIC).

## SAVING A COMPILED PROGRAM

You can use SAVE, LOAD, or RUN, in the normal way. CSAVE, CLOAD? and CLOAD may be used on a compiled program under ACCEL3, but not under ACCEL4.

SAVE "SAMPLE/ACC" will save the compiled SAMPLE program.
LOAD "SAMPLE/ACC" will reload it.
RUN "SAMPLE/ACC" will load and run it.

1)You must have ACCEL3/4 active to SAVE, LOAD, or RUN a compiled program, (i.e. you must have issued CMD"I","ACCEL" earlier).

2)You must have exactly the same environment in effect when you reload a program, as when you saved it. You must be running under the same version of the operating system (TRSDOS, LDOS, etc.), and you must have specified the same number of disk I/O buffers. This restriction comes about because the compiled program contains absolute machine addresses, and it can only run at one location.

3)The source file of a program, and the SAVEd compiled program are two very different things. It's easy to inadvertently SAVE a compiled program using the same name as the source. If you do this, your source is lost for ever. As a discipline, use "PROG/BAS" for the source file, and "PROG/ACC" for the compiled file.

## COMPILER ACTIVATION

The TRS-80 Level2 code in ROM provides a table of transfer addresses through which flow passes at certain key points in execution. ACCEL3/4 uses 3 of these to get control:

1) At the beginning of execution of each program statement.
2) At the beginning of execution of each direct command.
3) After execution of RUN, NEW, CLEAR, LOAD, and END.

When you initialise DISK BASIC, ACCEL3/4 is not "known" by the system, and has no way of getting control. When you execute CMD"I","ACCEL" you cause the module ACCEL/CMD to be invoked. As well as invoking the modules that do the actual compilation, this module enables these traps by putting its own addresses in the transfer slots. Because ACCEL3/4 then gets control on each command or statement, it is able to support a new command /FIX, giving you a shorthand way of invoking the compiler on subsequent occasions. If you invoke ACCEL/CMD when the current BASIC program is null, then it will perform this activation only, and will not invoke the compiler proper.

For ACCEL4, activation results in the creation of a resident control routine of about 150 bytes. This is placed at the top of free memory, and the free memory pointer is adjusted accordingly. This is transparent in normal running. However you may be using other machine-code programs that play similar tricks, and these may interfere with ACCEL4, and it may become deactivated. It is also deactivated by returning to DOS and re-entering BASIC.

Deactivation, or simply failure to remember to activate the compiler, means that /FIX is diagnosed as a SYNTAX ERROR. Less obviously, a compiled program (loaded from disk for example) will fail to run - it acts like a null program. Simply activate ACCEL4 (by CMD"I","ACCEL") and then run the program. You dont have to reload it. Multiple reactivation does not recreate the resident control routine unless some other program has moved the free memory pointer in the meantime.

There are 4 modules which make up ACCEL4, used as follows:

| | |
|---|---|
| ACCEL/CMD | Initialisation stub, and "control" routine |
| ACCP1/CMD | Compiler, first pass |
| ACCP2/CMD | Compiler, second pass |
| ACCRT/CMD | Run-time routines (to interface with BASIC) |

ACCEL3 combines the above into the single module ACCEL3/CMD.

ACCSU/CMD is a special module that is only used when you want to invoke a compiled program directly from DOS, rather than DISK BASIC. It can be used with both ACCEL3 and ACCEL4.

The ACCEL3/4 modules are loaded under the normal TRSDOS rules, namely that drives are searched in numeric order for the module name. So you dont have to have the ACCEL3/4 modules on the system disk (or even all on the same disk). For ACCEL4, ACCP1 and ACCP2 overlay the area of memory used by DISK BASIC, and the last job of compilation is to re-establish the DISK BASIC environment. Under TRSDOS (but not LDOS) this has the side-effect of destroying any DEFUSR values that may have been in effect. (So a compiled program should set these up explicitly).

For ACCEL4, the runtime module, ACCRT/CMD, overlays the error transient area, and you will notice some immediate disk activity when you first RUN the program after compilation, and after you generate an error message. The runtime traps remain active, even if you subsequently run programs interpretively. ACCEL3/4 determines whether or not the resident program is compiled by looking for a first line consisting only of a single colon (:). So no source program may start in this way. When ACCEL3/4 gets control at the beginning of a BASIC statement, the decision to execute in-line code, rather than to leave a statement to the interpreter is based on detection of a colon, followed by line-end. Once ACCEL3/4 has made the switch to in-line code, this code runs uninterpreted through one or more statements or lines, until the next uncompiled statement is encountered. INTEGER operations, GOTO, GOSUB, and RETURN are uninterruptable, except by reboot. However, non-integer assign, NEXT, array referencing, SET, RESET, POINT, and PRINT, all contain a "fast" test for the BREAK key. This throws execution back to READY, and the program is not CONTinuable.

## INVOKING A COMPILED PROGRAM FROM UNDER DOS

If you have SAVEd a compiled program in an earlier session, you may wish to power-up and go straight into executing that compiled program, without manually activating DISK BASIC. In fact a compiled program cannot run under the pure DOS environment, and the module ACCSU/CMD is provided to automate this bring-up process. Suppose you saved the compiled form of the SAMPLE program as SAMPLE/ACC. The operations that need to be executed from a cold start (i.e. power-up into DOS) are:

```
BASIC              (or LBASIC, etc., under LDOS)
n                  (number of files)
m                  (protect memory, if necessary)
CMD"I","ACCEL"     (activate ACCEL3/4 by compiling the null program)
RUN"SAMPLE/ACC"
```

ACCSU provides you with a way of executing such a list of commands. Under DOS type ACCSU, to invoke set-up. You will then be prompted to enter a list of commands, a line at a time, terminated by CLEAR. Type in your own bring-up list, for your particular program. After exiting with CLEAR your commands will be encapsulated in the in-core image of ACCSU/CMD, which resides at locations X'7000' and up. So DUMP this core image as a command file, e.g.

```
DUMP SAMPLE/CMD (START=X'7000',END=X'7100',TRA=X'7000')
(Note that the syntax of DUMP differs between TRSDOS and LDOS, and Model I and Model III)
```

Now, next time you power up, simply type SAMPLE. This will invoke SAMPLE/CMD, will execute the encapsulated commands, and will run the SAMPLE program, "directly" from DOS.

Warnings:

1)The module you DUMP must be large enough to contain all the commands you typed. The END address, X'7100', allows for over 100 bytes, which will cover all normal requirements, but if you use a long command list you may need to increase the DUMPed size.

2)DUMP is not supported on smal-LDOS, but smal-LDOS will run a module created on TRSDOS or LDOS.

## CHAINING PROGRAMS FROM DISK

ACCEL3/4 allows you to chain programs together, i.e. to proceed through a sequence of routines, each invoking the next from disk, and being overlaid by it. (The LDOS option which preserves variable values across RUN is not supported.). The chained programs may be either compiled or interpreted, or a mixture. You will need to debug these segments in an arbitrary order, compiling each one after it is checked out, and you will not want to change the chaining program, when the program it chains to is compiled.

The best way to achieve this is as follows. Adopt the convention that source programs are named e.g. PROG1/BAS, while the compiled version of the same program is PROG1/ACC. Since you want your final set-up to run compiled, use RUN "PROG1/ACC" etc. anywhere a chaining statement appears in a program. While debugging, simply store a double copy of each source program, one as BAS and one as ACC. So initially the whole system runs uncompiled. Now, when PROG1 is debugged, save its compiled version over the top of PROG1/ACC. Your total system will run as a mixture of compiled and uncompiled routines, while you gradually check out and compile the various sections.

## SELLING COMPILED PROGRAMS

One of the major attractions of a BASIC compiler is that it enables you to write BASIC programs for sale which, with care and tuning, can be comparable in performance with machine-code programs. Secondly, and no less important, a compiled program is very difficult to steal. It can be copied, of course, since any file can be copied byte for byte, but it cannot be modified, except by the owner of the original source BASIC. And you dont have to release this when you sell a compiled program.

To produce a disk for sale (or as a gift) to run the compiled version of SAMPLE, you must not only include SAMPLE/ACC (the compiled program), and SAMPLE/CMD (or its equivalent) but also the compiler control and run-time routines in the package. For ACCEL4 these are the modules ACCEL/CMD and ACCRT/CMD, as provided on the product disk. For ACCEL3 you need the first 1536 bytes of the total installed product. For simplicity these bytes have been packaged as a separate unit, which you can optionally install, and which is given the name ACC3RT/CMD. This routine MUST be located at the same address as the total ACCEL3 product.

The run-time routines are copyright modules, but their resale will be permitted, provided

1) You do NOT include either of the two compiler modules, ACCP1/CMD and ACCP2/CMD, or the body of the ACCEL3 compiler, on the sale disk, or in any way permit them to be used by a third party.

2) You give an acknowledgement in your program documentation that the ACCEL3 or ACCEL4 compiler was used.

Since the compiled program will only run under the same environment as it was compiled, it is strongly recommended that you supply an encapsulated command (e.g. SAMPLE/CMD) along with the product.

## EXECUTION PERFORMANCE

The aim of using a compiler is to improve execution speed. But the compiler cannot do better than the machine on which the program runs. The Z80 CPU chip is remarkably cheap, reliable, and fast, but it lacks many common operations (such as multiply and divide). These have to be executed via calls to ROM routines which provide the required function (e.g. multiply by successive additions), and this is of course relatively slow. The complex table at the end of this section is a guide to what features can be improved by compilation, and by how much. It remains one of the programmer's tasks (unfortunately), to match the requirements of the problem to the capabilities of the underlying computing system. The extra effort needed to optimise performance could be thought of as a form of machine-code programming. It can produce results comparable in performance with real assembler language coding, but it is incomparably easier, because debugging is in BASIC, using PRINT statements, TRACE, etc.

The result of compilation is a program which is a mixture of BASIC statements and directly executing Z80 machine-code instructions. The Z80 can execute branches and subroutine calls, and can perform logic and arithmetic (excluding multiply and divide) on INTEGERs, but not on SINGLE or DOUBLE precision floating-point numbers. Nor can it directly manipulate the internal form of BASIC strings, although it can move strips of bytes from one variable to another quite efficiently. (The difficulty with strings is that their lengths vary dynamically). So ACCEL3/4 translates many statements to sequences of calls to routines in ROM, or to its own run-time component.

In addition to the actual execution of the program operations, there is the "resolution" of the variable names and line-numbers. Here a compiler comes into its own. The BASIC interpreter resolves each name by a sequential search through its dictionary (table of variables), every time the variable is referenced during execution. In contrast the compiler allocates storage for the variable once during compilation, and then replaces each compiled reference by a direct machine address, rather than a dictionary search. Similarly each reference to a line number in GOTO or GOSUB translates to a simple branch address, whereas the BASIC interpreter has to search the program sequentially from the top to find the target line.

One effect of BASIC's two forms of sequential search is that the running time of a program depends on how large it is. The more variables you have in your program, then the longer the average time taken to find each one, and the more lines in your program, the longer it takes to execute each GOTO or GOSUB. The speed of the compiled code, on the other hand, is independent of program size and number of variables. This means that it is quite impossible ever to make a firm statement about relative performance, since you cannot say how long a statement such as A = B + C will take under the interpreter. It depends on context. Similar arguments apply to program size before and after compilation. Programs may contain REMarks and blanks. BASIC names can be any length. After compilation all these uncertainties disappear - the REMarks and blanks are removed and the variable and line references (in translated code) are all two-byte addresses.

So the table that follows is in one sense very pessimistic. The timings were all taken on the smallest program in which they could be measured, i.e. a simple FOR-loop. There were no blanks or remarks in the source, and the names were all two bytes long. The performance improvement measured for GOTO, for example, is 216 to 1. In a large program this would be even greater. But the catch is that this figure may be irrelevant. Because the directly executing operations are so fast, they may scarcely contribute to the execution total at all, and performance becomes dominated by those operations that are not compiled, e.g. READ, by the out-of-line subroutines, e.g. Multiply, or by I/O.

## SPEED/SPACE Performance Table

| Speed Improvement(Ratio) | | | | Operation | Space Degradation(Bytes) | | | |
|---|---|---|---|---|---|---|---|---|
| INT | SNG | DBL | STR | | INT | SNG | DBL | STR |
| 178 | 28 | 20 | 7.3 | Assignment (LET) | -4 | 0 | 0 | 0 |
| 3.5 | 3.6 | 3.6 | 3.5 | Array Reference (1-dim) | 13 | 13 | 13 | 13 |
| 3.0 | 3.0 | 3.0 | 3.0 | Array Reference (2-dim) | 12 | 12 | 12 | 12 |
| 35 | 1.8 | 1.6 | | AND, OR | 4 | 7 | 7 | |
| 23 | 2.0 | 1.6 | 6.6 | Compare (=) | 3 | 10 | 10 | 3 |
| 57 | 1.8 | 1.4 | 3.6 | Add, Concatenate (+) | 1 | 6 | 6 | 2 |
| 48 | 1.8 | 1.3 | | Subtract (-) | 4 | 6 | 6 | |
| 1.5 | 1.5 | 1.1 | | Multiply (*) | 6 | 6 | 6 | |
| 1.08 | 1.17 | 1.02 | | Divide (/) | 6 | 6 | 6 | |
| 77 | 70 | 84 | 9.3 | Constant Reference | 0 | 6 | 4 | 4 |
| 7.1 | 1.9 | | | FOR-NEXT | 6 | 23 | | |
| 111 | 6.8 | 4.8 | | POKE | -1 | 5 | 5 | |
| 10 | 4.5 | 3.6 | | SET, RESET | -1 | 5 | 5 | |
| 47 | 4.6 | 3.0 | 8.1 | IF THEN ELSE | 3 | 9 | 9 | 3 |
| 33 | 4.3 | 3.5 | | ON expression GOTO | -2 | 0 | 0 | |
| 50 | 6.8 | 5.1 | | ON expression GOSUB | 0 | 3 | 3 | |
| 1.2 | 1.01 | 1.03 | 1.2 | PRINT simple-variable | -1 | -1 | -1 | -1 |
| 61 | 5.0 | 3.7 | | OUT | 5 | 11 | 11 | |
| | | | | Flow of Control | | | | |
| | 216 | | | GOTO | | -7 | | |
| | 74 | | | GOSUB/RETURN | | -10 | | |
| | | | | Functions | | | | |
| inf | inf | inf | inf | VARPTR | -3 | -3 | -3 | -3 |
| 5.2 | 1.9 | 1.7 | | POINT | 3 | 9 | 9 | |
| 38 | 2.3 | 2.0 | | INP | 5 | 8 | 8 | |
| 149 | 2.3 | 2.0 | | PEEK | 0 | 3 | 3 | |
| | | | | String Functions | | | | |
| | | | 53 | ASC | | | | 5 |
| | | | 258 | LEN | | | | 0 |
| | | | 4.8 | LEFT$ | | | | 1 |
| | | | 4.7 | RIGHT$ | | | | 1 |
| | | | 6.4 | MID$ | | | | 2 |
| | | | 25 | CHR$ | | | | 0 |
| | | | 36 | CVI | | | | 0 |
| | | | 16 | MKI$ | | | | 0 |
| | | | 7.1 | CVS | | | | 0 |
| | | | 25 | MKS$ | | | | 0 |
| | | | 5.4 | CVD | | | | 0 |
| | | | 16 | MKD$ | | | | 0 |

Disclaimers-

1) Absolutely no commitment is implied by these figures. They are subject to all sorts of variability. E.g. the time to reference a constant depends on the actual value of the constant.

2) The space figures are for each repeated instance of use of the function. A compiled program has an additional one-time overhead of about 30 bytes, and for ACCEL4 also a more significant "first-time" overhead for certain constructs, notably FOR-NEXT loops and STRING functions. E.g. the first occurrence in a program of an INTEGER FOR-NEXT loop will cause ACCEL4 to generate an in-line subroutine of over 100 bytes. Second and subsequent INTEGER FOR-NEXTs will reuse this subroutine, not regenerate it.

3) Speed ratios for STRING operations depend on the lengths of the strings, whether the string is a program constant (a literal), whether the receiving string is the same length as the source string, etc. In measurement 4-byte strings were used.

4) Use of "inf" (infinity) in the table means that the ratio could not be measured meaningfully. In a compiled program, the reference to VARPTR(X) is faster than the reference to X.

4) Negative numbers in the space table mean that the compiled code occupied less space than the original. These numbers are based on the assumption that one statement per line is used. GOTO5000 occupies 10 bytes in BASIC (5 for the line overhead, 1 for the GOTO keyword, and 4 for the line number). In compiled code this becomes a single 3-byte instruction.

## PROGRAMMING EXAMPLES

These examples illustrate specific advantages that can be achieved by compilation. The first program allows you to produce musical notes from a BASIC program via the tape output port. In its uncompiled form the program runs so slowly that the waveform generated sounds like a series of blips, much like a Geiger-counter. Compiled, a top note of two octaves above middle C (1024 cycles) is easily achieved.

The second example is much more worthwhile. Every business application involves some degree of validation of the keyed input. This validation has to meet two conflicting requirements. First, it must diagnose any detectable errors immediately, and request the operator to rekey. Second, although this validation code may be quite complex, it must not be so slow that it causes the loss of any operator keystrokes. Apart from introducing errors, this has the effect of causing the operator to stumble, and lose confidence. So in this second example we are not looking for start-to-end speed-ups as the result of compilation. Rather, we are looking for better human factors.

These sample programs are included on the product disk you received. If other samples are included, then their running instructions will appear as comments at head of the program.

(A) SINGLE-NOTE MUSIC MAKER

The tape output is port number 255. You can drive this from a BASIC program by the statement OUT 255,X where X is a value sent to the port. If the least significant bit of X is 0 then the tape signal latch goes low. If it is 1 then it goes high. So by driving it alternately high and low you can generate a square wave. The frequency of this wave will control the pitch you hear, if you put the signal through an audio amplifier. The length of the note is decided by how long you make the loop. The volume cannot be altered.

1)The tape output signal is on the larger grey jack on the TRS-80. On Video Genie, if you have no internal sound, then use the 2nd cassette port as output, and insert OUT 254,16 at the head of the program to switch to this port.

2)A square-wave makes a nasty "electronic" sound. You can improve the sweetness by putting it through a circuit with a poor response to high frequency, e.g. 2000 c/s maximum.

3)The delay values which control the pitch do not give a uniform table. This is because the inner loop has a non-linear overhead which itself depends on the frequency.

4)On the Model III the timer is not disabled by CMD"T". The timer interrupts produce a crackle, which can be eliminated by calling a USR routine to disable interrupts. This routine is two bytes long, X'F3' (disable) and X'C9' (return).

```
10 'SINGLE NOTE MUSIC MAKER, USING TAPE OUTPUT PORT
20 DEFINT A-Z
30 DIM P(100),L(100)
40 'PITCH OF NOTES, FOR 3 OCTAVES, 128 TO 1024 C/S APPROX.
50 'C   C♯  D   D♯  E   F   F♯  G   G♯  A   A♯  B   C
60 '284,268,250,236,223,208,196,183,172,161,152,144,134
70 '134,126,118,111,104, 97, 90, 85, 78, 73, 68, 63, 59
80 ' 59, 55, 51, 47, 43, 39, 36, 33, 31, 29, 26, 24, 21
90 READ N 'NUMBER OF NOTES TO PLAY
100 DATA 23
110 'TAKE A PAIR OF SPARKLING EYES
120 'PITCH OF NOTE, FOR THIS TUNE
130 DATA 134,104,85,73,63,59,59,63,73,85,85,73,104,85,97,104,118,97,97,104,118,134,134
140 'LENGTH OF NOTE PLAYED, IN QUAVERS. NEGATIVE MEANS REST
150 DATA 2,1,2,1,2,1,4,1,1,2,1,2,1,4,1,1,2,1,1,1,1,4,-2
160 FOR I=1 TO N:READ P(I):NEXT 'PITCHES
170 FOR I=1 TO N:READ L(I):NEXT 'LENGTHS
180 FOR CC=1 TO 2 'PLAY 2 PHRASES
190 FOR C=1 TO N 'PLAY N NOTES
200 LL=0:LM=L(C) 'LENGTH OF NOTE, IN QUAVERS
210 R=1:IF LM<0 THEN R=0:LM=-LM 'REST?
220 I=0:K=P(C):L=K:M=0
230 OUT 255,M 'OUTPUT SIGNAL, ODD=HIGH, EVEN=LOW
240 I=I+10:IF I<L THEN 240 'DELAY, TO PRODUCE PITCH
250 L=L+K:M=R-M 'SWITCH WAVEFORM SIGNAL
260 IF I<10000 THEN 230 'PLAY ONE QUAVER. (CHANGE THIS CONSTANT TO ALTER SPEED OF MUSIC)
270 LL=LL+1:IF LL<LM THEN 220 'ANOTHER QUAVER WITHIN THIS NOTE?
280 NEXT C 'NEXT NOTE
290 NEXT CC 'NEXT PHRASE
300 END
```

(B) INPUT VALIDATION

This program collects a Work Order specification for the hypothetical ACME freight company. The first input field is a customer account number, validated for length, for numerics only, and against a Modulus-11 check. When this program runs under interpretive BASIC, a very fast typist can exit from this field and lose the first character of the next field by keying the second character before the validation completes successfully.

The second field is a two-character US State code, checked against a table of the 50 possible values. This is a very common form of validation. Other examples are commodity type codes, insurance classifications, tax codings, etc. In this case, although an early code like CAlifornia causes no problems, a search for e.g. WYoming causes a visible delay, and 2 or 3 keystrokes can easily be lost from the next field. In effect the operator must stop and wait for validation to complete before keying the next field. Compilation solves this important human problem, although of course it makes little difference to the overall throughput.

```
10 'DATA VALIDATION EXAMPLE - FREIGHT ROUTING
20 CLEAR 1000
30 DEFSTR A-H,S-Z:DEFINT I-N
40 DIM SC(50) 'STATE CODE
50 GOSUB 330 'INITIALISE
60 CLS:PRINT @10,"ACME FREIGHT COMPANY - WORK ORDER"
70 PRINT @138,"ENTER CUSTOMER NUMBER: ";CHR$(30);
90 INPUT CUSTNO
100 IF LEN(CUSTNO)<>5 THEN PRINT @970,"CUSTOMER NUMBER NOT 5 DIGITS";CHR$(30);:GOTO 70
110 MODSUM=0
120 FOR I=1 TO 5
130 CN=MID$(CUSTNO,I,1)
140 IF CN<"0" OR CN>"9" THEN PRINT @970,"NON-NUMERIC DATA IN CUSTOMER NUMBER";CHR$(30);:GOTO 70
150 MODSUM=MODSUM+ASC(CN)-48 'COMPUTE MODULUS-11 CHECK
160 NEXT
170 IF 11*INT(MODSUM/11)<>MODSUM THEN PRINT @970,"CUSTOMER NUMBER FAILED MODULUS CHECK";CHR$(30);:GOTO 70
180 PRINT @970,CHR$(30); 'CLEAR ERROR MESSAGE, IF ANY
190 PRINT @266,"ENTER DESTINATION (STATE): ";CHR$(30);
210 INPUT STATE
220 FOR I=1 TO 50
230 IF STATE=SC(I) THEN GOTO 260 'FOUND IT
240 NEXT
250 PRINT @970,"INVALID STATE CODE";CHR$(30);:GOTO 190
260 PRINT @970,CHR$(30); 'CLEAR ERROR MESSAGE, IF ANY
270 PRINT @394,"ENTER GOODS CLASSIFICATION: ";
280 INPUT GOODS
290 PRINT @522,"END OF TEST CASE. HIT ENTER TO RERUN. ";
300 INPUT X:GOTO 60
330 'INITIALISATION
340 FOR I=1 TO 50
350 READ SC(I) 'READ IN STATE CODES
360 NEXT
370 RETURN
380 DATA AL,AR,AS,CA,CO,CN,DW,DC,FL,GA,HA,ID,IL,IN,IO,KA,KY,LA,ME,MD,MA,MI,MT,MN,MR,MT,NB,NV,NH,NJ,
NM,NY,NC,ND,OH,OK,OR,PA,RI,SC,SD,TN,TX,UT,VT,VA,WA,WV,WI,WY
```

## PERFORMANCE HINTS

Nothing the compiler can do will speed up I/O devices - disk, tape, printer, or keyboard. But for processing limited by computation the following are good rules:

1) Always use INTEGER data types whenever possible, since these are the only data elements the CPU can manipulate directly. You can qualify variable names with % to make them INTEGERs, but better is to get into the habit of coding e.g. DEFINT I-P at the head of each program.

2) Because FOR-NEXT processing has to be "defensive", in terms of handling badly-behaved loops, it transpires that a programmed loop (e.g. I=I+1:IF I<100 THEN GOTO n) is very much faster. So it may be worth using such a technique on critical inner loops.

3) Avoid continually processing DATA with READ statements. Rather, READ the data values once into an array and process from that. This avoids the very considerable overhead of converting the DATA constants from character to numeric on every use.

4) There is a well-known execution "hiccup" caused by string space "garbage collection", (recovery of free space). ACCEL3/4 does not affect the actual garbage collection process, but it does attempt to minimise its frequency of occurrence, by avoiding string space allocation if possible. In particular, if string sizes match in assignment, then a spectacular improvement may result.

5) Keystroke polling. The key overrun example earlier showed how it was possible for ACCEL3/4 to substantially improve the keying characteristics of a program, by reducing the processing time between INPUT statements. However there is one situation where the compiled program may appear to behave worse. Suppose you have a real-time simulation, such as a game like Space Invaders, where your program continually updates the screen and periodically polls the keyboard, using the INKEY$ function. If INKEY$ is null, you loop round and perform the next update. If this update is both long and fully compiled, then it is possible that the player may depress and release a key in between the INKEY$ polls. In this case the keystroke is lost. Interpretive BASIC reduces the chance of this by polling the keyboard at the beginning of every statement (whether or not it asks for input). The cost of this poll is high - in a graphics test case, putting the poll into compiled code actually slowed down the program by a factor of 3. So it is omitted from compiled code, but included in uncompiled statements. (In any case, it's not a perfect solution. Interpreted BASIC may also lose keystrokes). If you have a compiled program that you believe suffers from this problem, then precede some of the statements in the update loop with a colon (:), to force the poll to take place more often.

## THE NOEXPR OPTION

ACCEL3/4 supports a compile-time option which minimises the level of optimisation. Code the single line-

```
REM NOEXPR
```

in front of the section where optimisation is to be minimised. You can turn optimisation back on with:

```
REM EXPR
```

There are a number of reasons for using REM NOEXPR in front of part of a program:

1)If that section contains extended non-Tandy language, then this may be a way of having it execute successfully. In particular, application programs for the Electric NoteBook Database Manager use a form of FN extension which will only work correctly (in a compiled program) if protected by REM NOEXPR.

2)The section may make use of ON ERROR GOTO. This may fail in an optimised section because either the error may not be correctly diagnosed, or, if it is, the error line number may not be up-to-date, so RESUME will not work.

3)To minimise code expansion. Since code expansion is not great with ACCEL3/4, this use is less important than it was with ACCEL and ACCEL2. However, array references in particular give quite a lot of compiled code, and in a non-performance-critical section you may prefer to have these interpreted.

4)If the compiled program fails. This might be due to integer overflow, for example. Preceding the program with REM NOEXPR may either make it easier to trace by running with trace on (TRON), or may eliminate it, in which case it can be identified by limiting optimisation to a section at a time.

REM NOEXPR inhibits compilation of all statements except GOTO, GOSUB, RETURN, RESTORE, FOR, NEXT, ON expr, and IF THEN ELSE (although the statements after THEN and ELSE and the expressions after IF, FOR, and ON are uncompiled). All of the above have to be compiled for the program to retain its integrity.

It is inevitable that some non-Tandy language extensions will still fail. For instance GOSUB X, where X is a variable containing a dynamically varying line number, would not be understood by ACCEL3/4, and could not work. The extensions that function correctly rely on the fact that ACCEL3/4 will pass a string of unrecognised bytes to the interpreter. Since ACCEL3/4 maintains the run-time variable dictionary exactly as the interpreter expects, such statements or expressions will work, if that's all they depend on. But ACCEL3/4 does not maintain either the LINE structure of the program, nor the run-time stack, in a compatible form.

If ACCEL3/4 finds an unrecognised function reference within an expression it will pass just that reference to the interpreter. However it assumes that the data type resulting from that reference is SINGLE. This may be wrong, and in this case make sure the reference is protected with REM NOEXPR.

## COMMON PITFALLS

1) Many programs have loops that are simply there to delay the process, e.g. to make a "ball" moving on the screen go more slowly. Either lengthen these loops when the program is compiled, or use a FOR-LOOP containing a very slow operation, like DOUBLE divide, which will swamp the compile speed-up.

2) 100 GOTO 100 is a common way of terminating a program to avoid the READY message corrupting the screen. This loop cannot be interrupted by the BREAK key, and will need RESET. Instead use e.g. 100 :GOTO 100

3) When you have compiled a program, do not use the editing commands, since they will produce completely unpredictable results. Always reset the machine state with NEW, LOAD, CLOAD, or RUN"prog".

4) It is common practice to use DATA statements as a source of variable data. I.e. after running the program once you EDIT new values into the DATA statements for the next run. This isn't possible once the program is compiled. Instead you have to modify the source and recompile.

## COMPILE-TIME MESSAGES

These are messages you may get when compiling a program with ACCEL3/4.

| | | |
|---|---|---|
| OM | OUT OF MEMORY. | Compiler could not complete. |
| FC | ILLEGAL FUNCTION. | Disallowed statement, e.g. DELETE. |
| UL | UNDEFINED LINE. | Bad line number referenced in GOTO, GOSUB, RUN, etc. |
| SN | SYNTAX ERROR. | Compiler cant parse the line. |
| TM | TYPE MISMATCH. | STRING/numeric data mismatch. |
| MO | MISSING OPERAND. | Bad syntax. |
| ST | STRING FORMULA TOO COMPLEX. | ACCEL3/4 restriction. Break the statement down. |

Compilation will diagnose ALL undefined LINE references, even those never executed when the program was interpreted. This is a common shock.

The result of any error (except UNDEFINED LINE) found at compile-time will be to leave the program in an indeterminate state. Dont even attempt to LIST it. Note down the error line number, and reload the original.

During compilation 3 numbers are displayed. The first is the size (in bytes) of the original BASIC program. The next 2 are the sizes after the two compiler passes over the program.

PASS 1 builds the variable dictionary, and modifies some of the source statements, e.g. DATA statements are moved to the back. It removes REMarks and redundant blanks, so the program size will usually go down.

PASS 2 actually compiles the code, and is the one that expands the text.



## RESTRICTIONS

Experience of users of ACCEL and ACCEL2 showed that some programs working under BASIC failed in execution, or even in compilation. These failures were almost always due to the program infringing one or more of the restrictions below, rather than as a result of a compiler bug. So if you encounter a problem, believe that it is as a result of a restriction, and identify the problem by tracing the program, inserting diagnostic PRINT statements, or by breaking the program down into segments.

1) No redefinition of meaning of names.

The names in your program must mean the same whether the program is read globally as the compiler sees it, or executed dynamically, as the interpreter sees it. The ambiguity applies only to names that take the DEFined data type by default. Names like I% or S$(3) are always consistent. An example of a disallowed name is I=1:DEFINT I:I=1. The interpreter will treat the first I as SINGLE, and the second as INTEGER. The compiler will treat both as INTEGER, i.e. it sees DEFINT as applying to the whole program.

You are unlikely ever to do this sort of thing deliberately, but it can come about inadvertently if CLEAR is used other than at the top of the program. CLEAR resets variables types to default (SINGLE), and may therefore cause a variable to change from INTEGER to SINGLE without your meaning it to. A common error is-

```
10 DEFINT I-N
20 CLEAR 1000
```

2) Current line-number is not maintained.

Lines which start with statements that have been compiled to machine-code do not update the current line-number. Therefore BASIC diagnostic messages may be misleading. TRON will give an incomplete trace.

3) Error behaviour is not necessarily consistent.

Out-of-range arguments to string functions (e.g. MID$ offset and length) are rounded modulo 256. Values out-of-range in ON statements are treated as zero, not errors. Out-of-memory may not be diagnosed at run-time, and may cause a wild branch, or a reboot. Your program may contain errors which BASIC does not diagnose, but which the compiler will reject, for instance bad syntax in an ELSE clause which is never executed. Some error diagnosis will be imprecise, e.g. RETURN WITHOUT GOSUB is diagnosed as NEXT WITHOUT FOR. (Both are symptoms of an empty stack). IF (A=B) 100 is treated by the interpreter as IF (A=B) THEN GOTO 100. ACCEL3/4 cannot handle this, although it will accept IF expr THEN 100, IF expr GOTO 100, or IF expr PRINT A, etc.

INTEGER OVERFLOW is not necessarily diagnosed. It is rarely caused by addition or subtraction, but may come about through multiplication, which IS diagnosed, but possibly with the wrong line-number. E.g. A = PEEK(I) + 256 * PEEK(I+1) is typically used to calculate a STRING address, and will overflow if the address is in the upper half of memory, i.e. PEEK(I+1) is greater than 127. Correct the problem by forcing one of the arguments to be SINGLE, e.g. 256.0 * PEEK(I+1).

In general, programmed error handling (i.e. the use of ON ERROR) is suspect. This is firstly because the error you are trying to trap may not be caught by the compiled code at all. But secondly, even if the error is trapped, the current line number may be out-of-date, i.e. it is the last uncompiled line. So RESUME may cause a loop. Actually, this problem is not as severe as it sounds, because in practice ON ERROR GOTO is almost always used in conjunction with I/O statements to detect FILE NOT FOUND, DISK FULL, INPUT BEYOND END OF FILE, etc. Since I/O is not compiled, the error trap will work.

4) A first program line of a single colon is disallowed.

5) Compiled programs may not be EDITED.

When the machine holds a compiled program you may not use the commands EDIT, AUTO, DELETE, MERGE, and NAME, and obviously these must not appear in a program you try to compile. (This gives an ILLEGAL FUNCTION diagnostic). In addition GOSUB should not be used as a keyboard (i.e. direct) command.

6) Operational differences.

You cant arbitrarily GOTO or RUN any line of the compiled program, only those lines that haven't been optimised. (To force a line N to retain its BASIC line number, simply put RUN N or RESUME N somewhere harmless in the program). BREAK may "take" in an interpreted line, in which case CONTinue may work. Or BREAK may be detected by the ACCEL3/4 library, in which case control goes to READY. Or BREAK may not take at all, e.g. in a tight GOTO loop. Then you have to reboot. For ACCEL4 under TRSDOS (not LDOS) DEFUSR values are unset by compilation. The LDOS option RESTORE n (line number) is not supported.

7) Saving, Loading, and Running compiled programs.

Compiled programs contain address references to both variables and to code. These will only work if the program is reloaded at exactly the same address. In effect always use the same environment as when the program was saved.

CSAVE and CLOAD of compiled programs are supported by ACCEL3 but not by ACCEL4. The LDOS option on RUN "prog" to pass current variable values is not supported. SAVE (and CSAVE) can only be used in direct mode and may not appear in a compiled program. SAVE of a compiled program is only supported for a literal file name, e.g. SAVE "PROG". SAVE expr will not work. SAVE causes all variables to be cleared, and after LOAD you cannot LIST the program or execute GOTO to a line, until some other operation has been performed. For ACCEL3, owing to an incompatibility between NEWDOS80 and TRSDOS, a compiled program has to be specially reformatted before SAVE or CSAVE, and this gives a significant delay on a large program.

8) Complexity of STRING expressions.

ACCEL3/4 is more restrictive than the interpreter on how complex STRING expressions can be. This is diagnosed at compile-time, and if it occurs break the statement down into separate statements.

9) Keyboard Poll.

Compiled code does not poll the keyboard. This may cause different operator characteristics, for instance a delay in accepting a keystroke, or failure to pause a scrolled display. You can force the poll by inserting a colon at the front of a line.

## INSTALLING ACCEL3

This is an operation that normally only has to be done once. The ACCEL4 modules overlay specific areas of low memory. They should not and need not be relocated. But ACCEL3 must eventually run in PROTECTED memory - i.e. in that area set aside for machine-code programs, when you enter BASIC, by the answer to the MEMORY SIZE question. The shipped copy of ACCEL3 will run in a 48K machine, at the top of memory. It may be relocated to run on any size machine, or in conjunction with other machine-code programs which occupy protected memory.

Type RELOCATE ACCEL3/CMD (enter). This will display the program's current starting location, and its length, and will then prompt you for a new starting location. If you have a 48K machine, simply enter the current location, and remember its value, as the answer to the MEMORY SIZE? prompt when you enter Disk BASIC. If you have a smaller machine, or you have other programs in protected memory, then work out a new starting address by subtracting the program length from your highest free byte address. Remember this value as the answer to the MEMORY SIZE? prompt. (It is generally wise to leave the top 64 bytes of memory unused, to avoid the program being corrupted when you exit and enter Disk BASIC).

If you want to use ACCEL3 to compile programs for sale to a third party, then there are two more considerations:

1)To get the compiled program to run on a smaller machine than yours you may want to locate ACCEL3 lower in memory. Optimally you need only make room for the first 1536 bytes of ACCEL3 in your customer's machine, so relocate ACCEL3 at

32768-1536 = 31232 for a target 16K machine, or
49152-1536 = 47616 for a target 32K machine.

2)You will need to supply your customer with a module comprising just those 1536 bytes. Repeat the relocation process on ACC3RT/CMD. Type in the SAME relocation address as was used for the ACCEL3 full product (dont use ENTER). When you put this on your own sale disk you may choose to rename it as ACCEL/CMD or ACCEL3/CMD so that its invocation is consistent with that of the full compiler.

Since the bulk of this manual refers to ACCEL/CMD as the "compiler", you may find it easier to follow the manual for ACCEL3, if you rename ACCEL3/CMD to ACCEL/CMD. But obviously this can only be done if you dont want to use ACCEL4.

## OPERATING SYSTEM PECULIARITIES

1)TRSDOS has some bugs in the area of re-establishing the BASIC environment. After compiling with ACCEL4, issue RUN or CLEAR, rather than GOTO to a line in the compiled program.

2)Smal-LDOS destroys the BASIC environment when CMD"I","..." is executed. As a consequence you cant load and invoke ACCEL3 using CMD"I","ACCEL3". Instead, issue LOAD ACCEL3/CMD under smal-LDOS, enter BASIC, and then compile by BRANCHING to the first location of ACCEL3. (Subsequently /FIX may be used). Dont use the SYSTEM command to execute the branch since there is a bug in ROM which prevents this working under DISK BASIC. Instead define a USR value equal to the address of ACCEL3.

3)Although ACCEL4 can be invoked by CMD"I","ACCEL" under smal-LDOS (because it recreates the BASIC environment after compilation) it is not possible for it to preserve any existing trap addresses. So if you use the Southern Software EDITor (disk-overlay version) then you will find that it deactivates ACCEL4, and vice versa. You will have to use CMD"I","EDIT" and CMD"I","ACCEL" on each invocation instead of /EDIT and /FIX. Continual alternate reactivation in this way will create multiple resident control routines, so occasionally you should return to DOS and reenter BASIC to reclaim the lost storage.

4)NEWDOS and DOSPLUS preempt the use of / (slash). To use /FIX you will have to type blank/FIX.

5)NEWDOS does not support CMD"I","...". Instead type CMD"ACCEL3".

## SELLING ACCEL3-COMPILED PROGRAMS ON TAPE

Let's assume you want to compile a BASIC program and save it on tape in a form that can be loaded and run under Level2 (non-disk) BASIC by a user who does not own ACCEL3. CSAVE and CLOAD are insufficient, since the CLOADed program will only run if the ACCEL3 run-time routines are present. Instead you will need to dump 3 core image segments:-

1) Control storage, including program size, memory size, etc.
2) The program itself, including its dictionary of scalar (non-array) variables.
3) The ACCEL3 run-time routines which interface with interpretive BASIC.

To dump the core images you will need the Southern Software utility TSAVE (or one of many other tape utilities available).

1) Decide where you want the compiler to load, e.g. 31232 on a 16K target machine. Reinstall a copy of ACCEL3 at this address.

2) With this copy in core, or after loading a copy from disk, enter Level2 (using RESET with BREAK).

3) Load your BASIC program (from tape).

4) Compile it by BRANCHING to the first location of ACCEL3. E.g. if you have loaded ACCEL3 at 31232, then type:

```
SYSTEM (enter)
*? /31232 (enter)  (Branch to first byte of compiler).
program compiles...
READY
```

5) Load TSAVE, either into a separate area of protected memory, or on top of latter half of ACCEL3. Invoke it by branching to its first location.

6) Respond to the TSAVE prompts as follows:

```
FILENAME? MYPROG
RANGE? 16512,16863          (save control storage)
RANGE? 16548↑,16635↑        (save the compiled program)
RANGE? 31232,32767          (save the ACCEL3 run-time routines, i.e. start-addr to start-addr+1536)
RANGE? (enter)
START? 6681                 (dummy start address)
R                           (record)
rewind and check with C
```

Notes:

1)Locations 16512 to 16863 contain information such as program start and end addresses, dictionary size, MEMORY SIZE, etc. So when the tape is reloaded (using the SYSTEM command) MEMORY SIZE is automatically set. Also ACCEL3 is automatically activated.

2)16548↑ to 16635↑ means save the ranges defined by the values held in these locations. This includes the compiled program itself (however large it is), and the dictionary of scalar variables, but not the arrays.

3)To run the compiled program you must have the ACCEL3 run-time routines available, and in the same place as when the program was compiled. These routines constitute the first 1536 bytes of ACCEL3, so this range depends on where you have located the compiler. Do NOT save the whole compiler, or you will be infringing copyright.

4)On Video Genie use the ESCAPE key for upward arrow.

5)If you want your compiled program to run on both Model I and Model III then build the program on the Model III, and TSAVE it using the lower cassette rate. The BASIC program start address is higher for Model III. So the core-image from Model III will load on Model I, although the converse is not true. However it is vital that the byte the PRECEDES the program is a zero. So find the program start address from location 16548, subtract 1, and save an additional range consisting of just that one byte. Then when the program is reloaded the byte will be forced to zero.

The sale tape should be loaded under Level2 BASIC, using the SYSTEM command. An acknowledgement must be given in the program documentation that Southern Software's ACCEL3 was used in its production.